

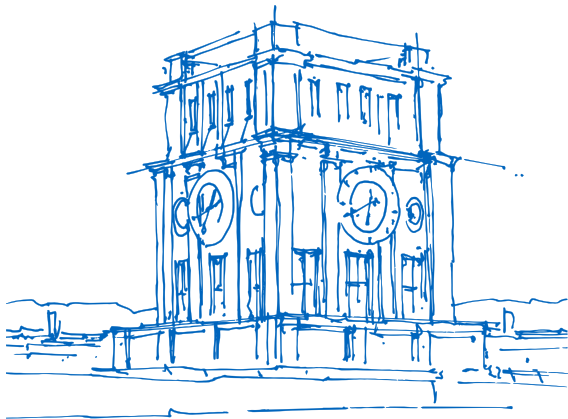
# Übung 03: RISC-V Deep Dive

## Einführung in die Rechnerarchitektur

**Michael Morandell**

School of Computation, Information and Technology  
Technische Universität München

04. – 10. November 2024



Montags:

<https://zulip.in.tum.de/#narrow/stream/2668-ERA-Tutorium—Mo-1000-4>



Donnerstags:

<https://zulip.in.tum.de/#narrow/stream/2657-ERA-Tutorium—Do-1200-2>



Website: <https://home.in.tum.de/momi/era/>

Keine Garantie für die Richtigkeit der Tutorfolien.  
Bei Unklarheiten/Unstimmigkeiten haben VL/ZÜ-Folien recht!

- Quiz
- Kurze Wiederholung
- Tutorblatt
  - ☐ Arrays und deren Adressierung
  - ☐ Zeichenketten/Strings
  - ☐ Taschenrechner-Tester (Präsenzaufgabe 01)

# Wiederholung

# Register & Calling Convention

- Argumente: a0 bis a5
- Rückgabewert: In a0 - a1 erwartet
- Temporäre Register: t0 - t6 können einfach überschrieben werden
- Saved Registers: s0 - s11 können genutzt werden, müssen vor return aber wiederhergestellt werden

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporary registers
s0/fp	x8	Saved register/Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments/Return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporary registers

# Wichtige Instruktionen

lw rd, imm(rs1)	load word	$rd = [Address]_{31:0}$
addi rd, rs1, imm	add immediate	$rd = rs1 + \text{SignExt}(imm)$
xor rd, rs1, rs2	xor	$rd = rs1 \wedge rs2$
slli rd, rs1, uimm	shift left logical immediate	$rd = rs1 \ll uimm$

sw rs2, imm(rs1)	store word	$[Address]_{31:0} = rs2$
------------------	------------	--------------------------

beq rs1, rs2, label	branch if =	if $(rs1 == rs2)$ $PC = BTA$
jalu rd, rs1, imm	jump and link register	$PC = rs1 + \text{SignExt}(imm)$ , $rd = PC + 4$

# If-else in RISC-V

## C Code

```
if (i == j)
    f = g + h;

f = f - i;
```

```
if (i == j)
    f = g + h;

else
    f = f - i;
```

## RISC-V Assembler

```
# s0 = f, s1 = g, s2 = h
# s3 = i, s4 = j
```

```
bne s3, s4, L1
add s0, s1, s2
L1:
sub s0, s0, s3
```

```
bne s3, s4, L1
add s0, s1, s2
j    done

L1:
sub s0, s0, s3
done:
```



# While in RISC-V

## C Code

```
// Berechne x, so dass  
//  $2^x = 128$ 
```

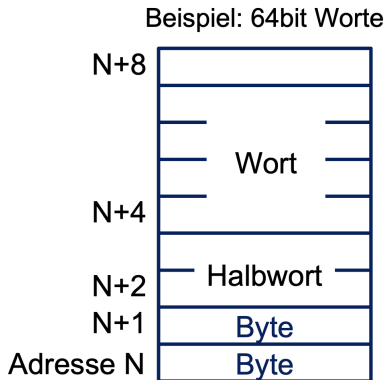
```
int pow = 1;  
int x   = 0;  
  
while (pow != 128) {  
    pow = pow * 2;  
    x = x + 1;  
}
```

## RISC-V Assembler

```
# s0 = pow, s1 = x
```

```
addi s0, zero, 1  
add  s1, zero, zero  
addi t0, zero, 128  
  
while:  
    beq s0, t0, done  
    slli s0, s0, 1  
    addi s1, s1, 1  
    j    while  
  
done:
```

- Speicher ist üblicherweise byte-adressierbar – bei uns auch
- Speicher unterteilt in Zellen; Zellengröße entspricht Verarbeitungsgröße



# Strings sind Arrays

- Strings sind nur Arrays aus Buchstaben
- Im Computer sind die Buchstaben durch Zahlen repräsentiert
- Zeichen sind ASCII enkodiert (0-127)
- 2 verschiedene Arten von Strings
  - ☐ C-Strings: Aneinanderreihung von Zeichen dann NULL-Byte
  - ☐ Pascal-Strings: Erst 1 Byte für die Länge des Strings, dann Zeichen

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

  Changed or added in 1963 version  
  Changed in both 1963 version and 1965 draft

# Structs

- Arrays für Daten gleiches Typs  $\Leftrightarrow$  structs für Daten unterschiedlichen Typs
- Assembler kennt keine Typen – nur Zahlen in verschiedenen Größen
- `la` = load address. Pseudoinstruktion, die Adresse in Register ladet

## RISC-V Assembler

example:

```
.word 0  
.short 0  
.word 0  
.byte 0
```

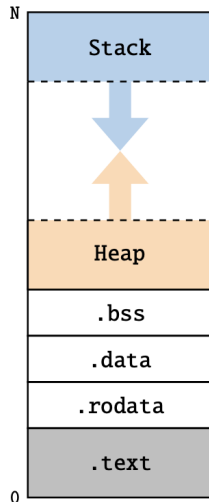
```
la a0, example  
li a1, 1  
sw a1, 6(a0)
```

## C Code

```
struct calc {  
    int operand1;  
    short operand2;  
    unsigned int ergebnis;  
    unsigned char operation;  
};  
struct calc example = { 0 };  
example.ergebnis = 1;
```

wichtige sog. Sections:

1. `.text`  
beinhaltet ausführbaren Code
2. `.data`  
beinhaltet globale, schreibbare, initialisierte Daten
3. `.rodata`  
beinhaltet globale, nicht-schreibbare, initialisierte Daten
4. `.bss` (= block starting symbol)  
beinhaltet globale, schreibbare, nicht-initialisierte Daten  
effektiv nur ein Marker für „hier kommen  $n$  Bytes“



# .data-Section in RISC-V

```
.org 0x200                // Boilerplate

.data                    // Start der .data Section

data_1: .word 1, 2, 3, 4  // Alloziert 4 Words (je 4 Byte) nacheinander (als Array)
                        // ab und initialisiert diese mit 1,2,3,4

text_1: .asciz "I <3 ERA!" // asciz = ASCII Zero -> fügt automatisch NULL-Byte an
                        // Alloziert 10 Bytes nacheinander und initialisiert diese
                        ↪ mit "I <3 ERA!" zzgl. NULL-Byte
                        // Erinnerung: Das Ende von Strings wird mit einem NULL-Byte
                        ↪ markiert
```

Fragen?

Bis zum nächsten Mal ;)

Folien inspiriert von Markus Gschoßmann und Clemens Schwarzmann